Title	近似代数計算のためのC++による数式処理クラスライブラリーの開発(数式処理における理論と応用の研究)
Author(s)	福井, 哲夫
Citation	数理解析研究所講究録 (1993), 848: 162-176
Issue Date	1993-09
URL	http://hdl.handle.net/2433/83653
Right	
Type	Departmental Bulletin Paper
Textversion	publisher

数理解析研究所講究録
第 84[illegible] 1993 年 162-176

近似代数計算のための C++ による 数式処理クラスライブラリーの開発

詫間電波工業高専　福井 哲夫 (Tetsuo FUKUI)

1. はじめに

近年、数値計算の高速性と数式処理の利点をうまく融合して、ある種の病的条件を含むような科学技術計算を効率よく処理するアルゴリズムが注目されてきている。これらは総称して近似代数計算と呼ばれ、その代表例として野田・佐々木氏の発表した近似 GCD 算法 [1][2] が挙げられる。しかし、数式処理の利用が複雑化してくるにつれて現在の代表的数式処理システムには次のような問題点が指摘される。

- 数値・数式融合処理に着目した場合、(1) 複数の言語を併用する必要があったり、(2) 一つの数式処理システムで可能な場合でも、インタプリタ型のため数値計算部分が大規模になると時間がかかりすぎる。
- 近似代数アルゴリズムの開発では途中の計算過程が重要であるが、システムの内部処理 (特に数値計算部) の挙動把握が困難である。
- 数式処理が必要なアプリケーション、例えば数式の必要な CAI プログラム等を考えた場合、数式処理をライブラリーのように道具として利用できるものが少ない (数式処理ライブラリーを提供しているすぐれたシステムとして、富士通国際研　野呂、竹島、加藤氏の開発した Risa [3][4][5] が挙げられる)。
- C 言語や FORTRAN のような処理系で記述された数式処理ライブラリーは一般に式の記述が醜くなる。

本論文ではそれらの問題点を解決する一つの試みとして、オブジェクト指向言語である C++ に着目する。C++ による数式処理クラスライブラリーのひな型を作り、それを 2 *cdot* 3 の例に応用してみることによって、数式処理ライブラリーを C++ で実現するための方法と利点について報告する。

ここでは C++ の優れた特徴 [6]-[9] について紹介する。C++ は基本的には拡張された C 言語と見ることが出来、構造体の取扱いが整備強化されている。この拡張された構造体のことをクラスと呼び、その構造体として宣言された変数のことをオブジェクトと呼んでいる。C++ の主な特徴を挙げると次のようになる。

- 演算子のオーバーロードができる。
  例えば、数式としての和の式を C 言語で記述する場合、多項式オブジェクトを x, y, z として、$x+y$ を z に代入させるには多項式和の関数 $addp()$ を作り、次のように

記述することになる。

$$addp(x,\ y,\ \&z); \tag{1}$$

これでは演算が複雑になってくると醜くなってしまう。一方、C++ では演算子'+'や'='を多項式クラスにも適用できるように追加定義できる。このことを演算子のオーバーロード (多重定義) と呼んでいる。これによって (1) と同様のことをさせるには次の様に記述すればよい。

$$z = x + y; \tag{2}$$

- 引数の型 (クラス) によって処理を分けることができる。
  例えば、2引数の和として多項式と数値および多項式と多項式では当然異なった処理が必要である。C 言語ではこれらの処理の振り分けは一つの関数内で動的に行われるが、C++ ではコンパイル時点で静的に振り分けることができる。
- クラス (構造体) の継承ができる。
  例えば、変数を扱うために変数名管理やメモリ管理機能を持った変数クラスが定義されていて、新たに多項式クラスを定義したい場合を考えよう。このためには変数クラスと同じ機能を持った主変数と係数指数リストの 2 つのメンバーを持ったクラス (構造体) を新たに作ればよい。C++ では変数クラスの機能をそのまま継承して、さらにメンバーを付け加えた新しいクラスを容易に作ることができる。したがって効率よく多項式クラスが定義できるのである。

以下では C++ で数式処理を実現するための方法、応用、利点と問題点を議論し、最後にむすびとする。

2. C++ から数式処理へのアプローチ

数式処理を C++ によって実現する方法を調べる上で、広範囲な数学が扱えるシステムを構築する必要はない。ここでは実数式の数式処理モデルを考え、まず C++ による数式処理のイメージを紹介し、数式処理ライブラリーの実現方法を探る。ここで言う実数式とは数，多項式，有理式，関数式の 4 つの多様性を持つ数式オブジェクト (Fig. 1) を意味する。実数式以外の数式表現は多数存在するが、例えば数成分ベクトルやそれ上の行列あるいは複素式等への表現上の発展は実数式の組として組織的に展開できるので、やはり実数式の実現方法が基本となる。

2.1. 数式処理モデル

数オブジェクトは次のような 3 つの理由から浮動小数点数のみとする。

(1) 組み込みの演算機能が利用できて簡単である。多倍長数などは数どうしの演算やメモリ管理を新たに作ることになるが、数式との関係は固定長の場合と同様に議論すればよい。

(2) 数値・数式融合処理への有効性を調べる上で数値計算との連携がスムーズである。

(3) 数値計算部分の挙動が処理系 C++ のもつ演算に帰着できるため明快となる。これは特に近似代数計算のアルゴリズムを調べる上で重要である。

実数式: Real
- 数 ・・・・ double (組み込み)
- 多項式 ・・ Polynomial (四則演算・微分・問い合わせ・変形)
- 有理式 ・・ Rational (四則演算)
- 関数式 ・・ Function (四則演算)

Fig. 1. 数式オブジェクト

数式処理機能としては実数式の四則演算、代入、出力、入力といった最小限のものとする。これらの処理はオブジェクトのクラス (数, 多項式, 有理式, 関数) を意識することなく行われるようにする。また、多項式については応用する上で必要なベキ演算、微分演算、簡単な問い合わせ、変形といった処理を付加する。

2.2. C++ による実行のイメージ

C++ による数式処理の実行は、定義済みの数式処理ライブラリーを使ったコンパイラ型言語によるプログラミングと同じである。以下ではこのような手続きによって数式処理を行うことを単に C++ 数式処理と呼ぶことにする。また、上記の数式処理モデルを C++ によって実現したライブラリーのことを数式処理クラスライブラリーと呼ぶことにする。

数式処理クラスライブラリーは数式クラスを定義したヘッダファイル ("appcalg.h") とクラスのメンバ関数を記述したライブラリーファイル ("appcalg.lib") の 2 つのファイルで構成される。現在のモデルにおいては実数式用のモジュールのみであるが、一般には様々な分野のモジュールの集合体として構成される。

この数式処理クラスライブラリーを使って数式処理プログラムを記述する一般的形式を Fig. 2 に示す。

```
#include "appcalg.h"
main()
{
  InitAppCAlg();      // C++ 数式処理の初期化
  数式処理コマンド
}
```

Fig. 2. C++ 数式処理のプログラム形式

初期化コマンド InitAppCAlg() によって外部変数等の準備が行われる。このコマンドはプログラムの初めに 1 度だけ実行すればよい。ヘッダファイル appcalg.h はカレントディレクトリーにあるものとしてコンパイルし、 appcalg.lib とリンクして実行される。このように、いたって C 言語の普通の形式として実現できる。

実数式の四則演算や代入は通常の数値変数とほぼ同様に記述できる。例えば実数式 x と y の和を z に代入するには (2) 式の様に記述すればよい。これは先に述べた演算子のオーバーロード機能を利用している。本モデルにおいて新たにオーバーロードされた演算子は 8 つある。それらの記号と意味を Fig. 3 に示す。

演算子記号	+	−	*	/
本来の機能	数値加算	数値減算 数値負号	数値乗算	数値割算
多重定義された意味	数式加算	数式減算 数式負号	数式乗算	数式割算
演算子記号	=	<<	>>	^
本来の機能	数値代入	数値・文字列出力	数値・文字列入力	排他的 OR
多重定義された意味	数式代入	数式出力	数式入力	数式ベキ演算

Fig. 3. オーバーロードされた演算子とその意味

```
#include "appcalg.h"
main()
{
  InitAppCAlg();                                  // 初期化
  Polynomial x("x"), y("y");                      // 多項式型変数 x, y の宣言
  Polynomial z, w;                                // ゼロ多項式型の宣言
  z=x+y-1;                                        // 多項式の計算と代入
  w=(z^3);                                        // べき計算
  cout<<"w = (x+y-1)^3\n  = "<<w<<"\n";           // 多項式のコンソール出力
  cout<<"diff(w,x) = "<<diff(w,x)<<"\n";          // x による微分
  cout<<"diff(w,x)-diff(w,y) = "
    <<diff(w,x)-diff(w,y)<<"\n";                  // x および y の微分の差
}
```

Fig. 4. サンプルプログラム

Fig. 4 は C++ 数式処理プログラムの簡単なサンプルである。プログラムでは多項式オブジェクト x, y, z, w が宣言子 Polynomial によって宣言される。初期値に文字列を指定すると、その文字列を変数名とする単項式が生成され、省略するとゼロ多項式が生成される。処理内容は $x+y-1$ の 3 乗を w とし、w 自身と w の x による微分および w の x, y による微分の差を計算出力している。微分はライブラリーに定義されている関数 diff() を使って行われ、式の出力は一般に cout << 式; という形式で実行される。Fig. 5 にサンプルの実行結果を示す。

```
w = (x+y-1)^3
  = [x^3+[3*y-3]*x^2+[3*y^2-6*y+3]*x+[y^3-3*y^2+3*y-1]]
diff(w,x) = [3*x^2+[6*y-6]*x+[3*y^2-6*y+3]]
diff(w,x)-diff(w,y) = 0
```

Fig. 5. サンプルプログラムの実行結果

3. C++ 数式処理の実現法

ここでは上記の数式処理モデルを C++ 上で実現するための方法について議論する。

3.1. 数式表現

まず、4 つの数式オブジェクトを C++ 上で表現するための方針について述べる。

数の表現は C++ に標準に定義されている倍精度 (double) 型浮動小数点数クラスを基本とし、多項式としての数および実数式としての数の 3 つの形態を取る。いずれもオブジェクトの一部に double 型数を含む構造としている。

多項式の表現としては再起的表現 [10] を採用する。再起的表現は REDUCE や MACSYMA などで使われている表現で、多項式を主変数とその指数・係数多項式のリストの組み合わせで構成する。例えば多項式のデータ構造を図示すると次のようになる。

主変数	x		
指数	2	1	0
係数多項式	$y+1$	3	$y+2$

Fig. 6. 多項式のデータ構造例

この表現の長所は、表現が一意的となり単純であるため多くの数学的アルゴリズムが適用し易い。しかし、式の次数によっては表現が大きくなりすぎたり、主変数の順序を管理しなければならないという欠点を持つ。

変数の順序は外部変数であるリストとして管理する。リストのノードは変数名とその変数の相対的位置を知るための順位をメンバーとする。このモデルでは順位の決定は、変数が宣言された順番に付けられ、変数名リストに追加されるようにした。

有理式の表現は最も単純な、分子多項式および分母多項式の対形式を採る。また、一意性のために reduced フラグを設け、分子分母が互いに素で、分母が正規化されていれば reduced=TRUE とする。

関数については簡単のため一変数関数を扱うこととし、関数と引数の組を一つの主変数とみなした多項式の構造と同じデータ構造を採用する。この構造の利点は表現が一意的となることと、多項式の処理で使われた四則演算等の手続きが全く平行に記述できることにある。しかし、例えば因数分解した表現の様に多彩な表現ができないという問題を残している。

C++ では多項式、有理式、関数式は次の 2 つの理由から、みな独立したデータ型として扱う方がよい。

(1) コンパイラ型プログラミングの利点の一つは必要なモジュールのみをリンクし、プログラムの実行コードサイズをより小さくできる点にある。したがって、一つのまとまったクラスをできるだけ一つのモジュールとして扱えるようにしたい。

(2) C++ 数式処理によって効率よくプログラミングするにはクラスの継承を利用して、既に有る資産を活用する方がよい。したがって、できるだけ基本的なクラスの利用可能な自由度を広げておく方がよい。

3.2. クラスの定義と静的関係

数式処理モデルを実現するために、Fig. 7 に示すような 5 つのクラスと 4 つの構造体を定義した。構造体は文法的には全てのメンバーが public なクラスを意味する。このモデルには 4 種類のリストが存在し、それらの構成要素 (ノード) は他のクラスからアクセスされるので構造体とした。多項式クラスが変数クラスのサブクラスとなっている理由は、変数クラスだけで独立した利用が必要なことと、変数クラスを継承できるように可能性を広げるためである。このように C++ ではクラスのメンバーに対して構造的に部分メンバーによる独立クラスを考える場合、通常の親子関係とは逆で、部分メンバークラスを親として基のクラスを子とする必要がある。関数クラスと実数式クラスも同様の関係となっている。

3.3. データ構造の動的関係

3.3.1. 外部変数

外部変数として変数名リストへのポインター RootVar と関数名リストへのポインター RootFun が用意されている。ただし、ユーザが直接このポインターをアクセスすることはなく、実数式オブジェクトの生成時に名前がリストへ追加されていく。変数名リストの先頭要素はヌル名であって数オブジェクトを意味し、他の変数と区別される。関数名リストも同様に先頭が数を意味するほか、第 2, 第 3 要素は多項式, 有理式として関数と区別される。例えば、Fig. 8 では変数名リストには “x” と “y” が登録され、関数名としては “f” が登録されている。

3.3.2. 数式クラスの多様性

多項式オブジェクトの特別な場合として数が含まれる。多項式の構造を持つ数とは主変数が RootVar を指し、係数指数リストを数値の意味に替えて表現する。例えば、値が 1 の多項式は Fig. 8 のオブジェクト c1 の構造となる。実数式の多様性についても同様の扱いがなされている。この方法の問題点は処理が複雑になることである。Fig. 8 では多項式オブジェクトとが生成されており、実数式としておよび関数が生成されている。

3.4. メモリー管理

式オブジェクトのメモリーへの割当および解放は C++ の new および delete コマンド [7] によって行う。これらは C 言語の malloc および free コマンドに相当するが、C++ の機能によって式オブジェクトの動的メンバーも含めてきめの細かい管理ができるようになっている。オブジェクトを解放するタイミングについては、このモデルでは演算実行時に式オブジェクトの temporary フラグが ON のときに解放するようにした。

＜クラス＞

Variable (変数クラス)

VarNode*	mainVar

Rational (有理式クラス)

int	reduced
Polynomial*	num
Polynomial*	den
int	temporary

Function (関数クラス)

FunNode*	mainFun

＜サブクラス＞

Polynomial (多項式クラス)

VarNode*	mainVar
CoefExp*	celist
int	temporary

Real (実数式クラス)

FunNode*	mainFun
Void*	args
RealCoef*	celist
int	temporary

＜構造体＞

VarNode (変数名リスト用ノード)

char*	name
int	order
VarNode*	next

CoefExp (多項式係数指数リスト用ノード)

Polynomial*	c
double	e
CoefExp*	next

FunNode (関数名リスト用ノード)

char*	name
int	argnum
int	order
FunNode*	next

RealCoef (実数式係数指数リスト用ノード)

Real*	c
double	e
RealCoef*	next

Fig. 7. 静的クラスの定義

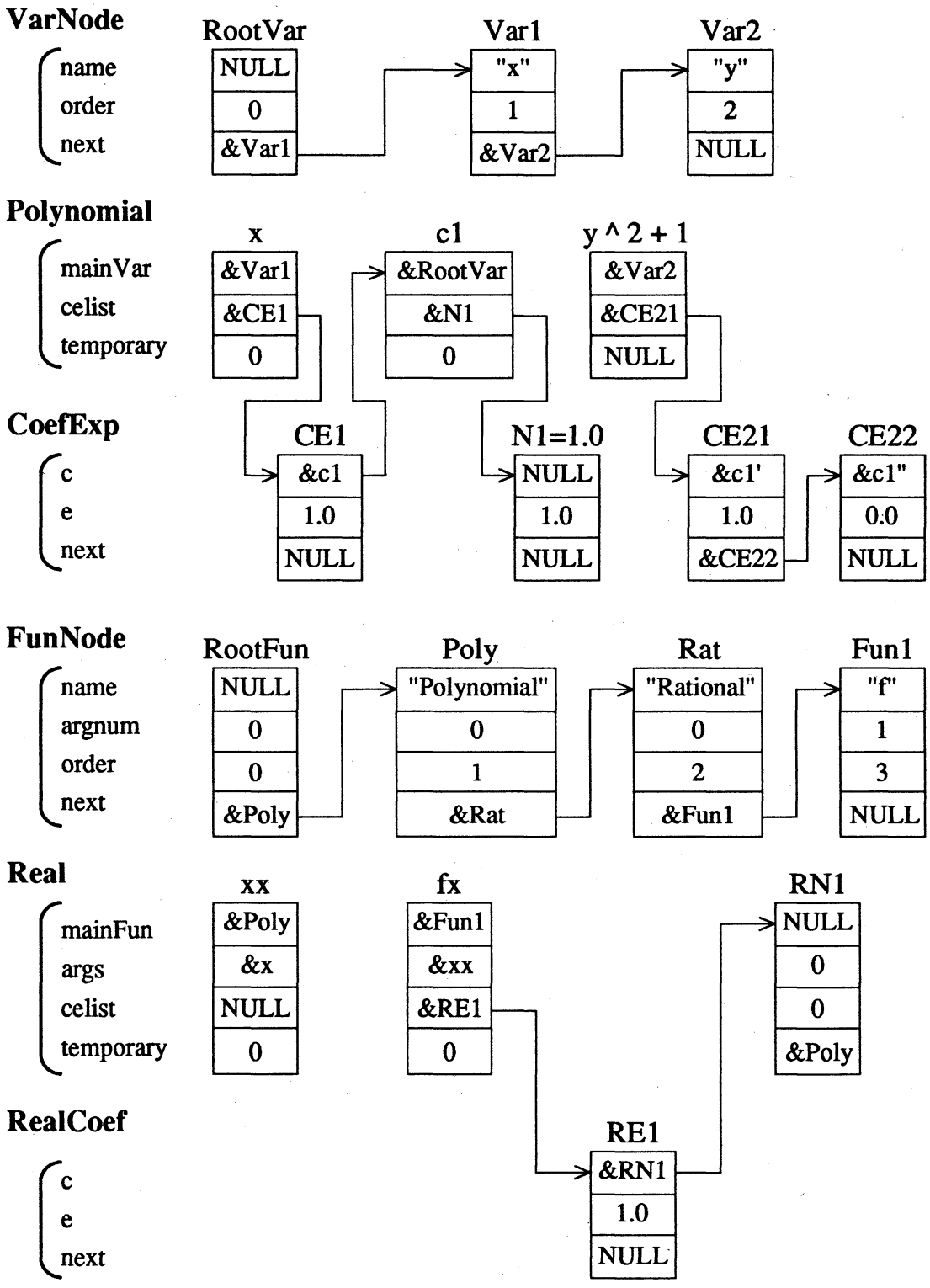

Fig. 8. データ構造の動的関係例

4. C++ 数式処理のサンプルと評価

ここでは4つのサンプルプログラムを紹介し、C++ 数式処理の特性を評価する。

4.1. ユーザ定義関数の例

実数式から実数式への sin 関数を定義した例を Fig. 9 に示す。処理内容は仮引数 x に実数式を受け取り、x が数なら数値 $\sin(x)$ なる実数式を、そうでなければ関数表現 $\sin(x)$ を返すものである。このように実数式を処理するためのユーザ定義関数が C++ の文法に従って容易に構築できる。このとき重要なことは関数に実数式引数を渡し、関数から実数式を返す手続きである。方法としては式オブジェクトへのポインターで渡す方法と式オブジェクトをコピーする方法が考えられる。前者の方法は簡単で高速に処理できる利点を持つが、不注意にデータを壊す危険性を持つため内部情報を持たないユーザには不向きである。C++ では次に挙げる機能によってユーザが意識することなく関数へ実数式の授受ができるようになっている。

- 関数にあるクラスの引数を渡したり、関数から返される際に、そのクラスに決められたコピー手続きが自動的に呼び出される。しかもこのコピー手続きはクラスのメンバーがポインターであってもそれが指示するオブジェクトも含めてコピーできるように設計できる。

```
Real sin(Real x)                          // sin 関数の定義
{
  if(x.is_Number())                       // x が数ならば
    return(Real(sin(x.s_value())));       // 数値 sin(x) を返す
  else                                    // そうでなければ
    return(Real("sin", x));               // 関数式 sin(x) を返す
}
```

Fig. 9. ユーザ定義関数のサンプル

```
#include "appcalg.h"
main()
{
  double n;
  InitAppCAlg();
  Polynomial x("x"), y("y");
  cout << "(x+y)^n :: n = ? ";cin >>n; cout << "\n\n";
  cout << ((x+y)^n) <<"\n";
}
```

Fig. 10. 2項ベキ計算のテストプログラム

Table 1. 2項ベキ計算の動作比較

n	(1) REDUCE 3.2	(2) Petit–Asir 0.4	(3) APPCALG.h 0.4
100	22.0 sec	4.4 sec	7.6 sec
200	85.3 sec	18.5 sec	24.7 sec
300	184.9 sec	40.5 sec	50.6 sec
400	359.2 sec	73.5 sec	86.2 sec
500	No free space	Stack overflow	131.0 sec
600			185.6 sec
700			249.0 sec
800			322.0 sec
900			×
各計測方法	起動最初の実行 1:(x+y)**400; から次のプロンプトまで 2:	起動最初の実行 [0](x+y)^400; から次のプロンプトまで [1]	プログラム起動後、n の入力からDOSプロンプトまで A:

4.2. 実行コードの効率比較例

ここで報告している C++ 数式処理のモデルでは数オブジェクトが固定長であること、必要なモジュールのみリンクできることなどからそれらの特徴を簡単なテストで比較してみよう。テストの内容は2項のベキ計算を $n = 100, 200, 300, \cdots$ について実行したものである。テストに使ったプログラムを Fig. 10 に示す。これを数式処理クラスライブラリーとコンパイルリンクした実行コードサイズは 99kB であった。本テストプログラムを含め次の3つのシステムで動作比較を行った。

(1) REDUCE Ver.3.2 / AMI–LISP Ver.1986.05.06
by 山本 強 (北大)
コードサイズ: 510kB (スタック: 8kB)

(2) Risa / Petit–Asir Ver.0.4
by 加藤昭彦, 野呂正行, 竹島卓 (富士通国際研)
コードサイズ: 547kB

(3) APPCALG.H Ver.0.4 ／ C++ 数式処理クラスライブラリー
by 福井哲夫 (詫間電波高専)
実行プログラムサイズ: 99kB

動作比較環境はすべて同じマシン
NEC 製パーソナルコンピュータ PC–9801RX
CPU: 80286 (12MHz クロック)
OS: MS–DOS Ver. 2.11 (フリーメモリー約 594kB)

```
問題：右の 2 次式を因数分解せよ。    =>  [x^2-6*x-7]
答を入力して下さい。                 =   (x-7)*(x+1)
★★★    正解です。    ★★★

問題：右の 2 次式を因数分解せよ。    =>  [x^2+13*x+30]
答を入力して下さい。                 =   (x+5)*(x+6)
× 正解は: [x+3][x+10] です。

問題：右の 2 次式を因数分解せよ。    =>  [x^2-4*x+3]
答を入力して下さい。                 =   (x-3)*(x-1)
★★★    正解です。    ★★★

問題：右の 2 次式を因数分解せよ。    =>  [x^2-3*x+2]
答を入力して下さい。                 =   ^Z
よくがんばりました。
```

Fig. 11. 因数分解練習ドリル実行例

でテストしている。各 n に対する動作時間と計測方法を表 1 に示す。このデータだけで評価することはできないが、処理速度については (2) の Risa/Petit-Asir Ver. 0.4 が優秀である。(1) より (3) が速くなっているのは第 1 に式の係数の値が (1) では多倍長整数であるのに対し、(3) では固定長であることが挙げられる。しかし、表示方法の違いなどがあるため単純な比較はできない。ただ、(3) の優れた点として、他が $n = 400$ までしかできなかったのに対して $n = 800$ まで実行できたことが挙げられる[1)]。これはコードサイズが他のインタプリタ型システムに比べて約 5 分の 1 の大きさであり、作業領域が広いためである。特に (3) のプログラムが小さくできたのは、C++ 数式処理モデルがまだ試行的とはいえ、必要な多項式の演算モジュールのみリンクできるためである。

4.3. 道具としての数式処理の例

ここでは CAI への応用、特に因数分解練習ドリルの例を紹介する。処理の内容は適当な x の 2 次式 $x^2 + Ax + B$ を作成し、学習者にその因数分解した式を答えさせるものである。まず、実行例を Fig. 11 に示す。出題された式に対して答を入力し、正しければその旨が、間違っていれば正しい答が表示されてまた出題から繰り返す。このようなプログラムは C++ 数式処理クラスライブラリーを使えば容易に作れる。プログラム例を Fig. 12 に示す。2 次式作成のための乱数の発生は C 言語の持つ rand() 関数を使用している。2 次式の生成、出題、入力、正誤判定については数式処理が必要な部分である。このように目的は CAI であるが道具として数式処理が必要な場合には C++ 数式処理クラスライブラリーが有効である。特に目的がグラフィックスや特殊なハードウェアを必要とするよう

[1)] 比較に用いた Risa/Petit-Asir システムは Risa/Asir の 640kB MS-DOS 限定版である。したがって、このデータは Risa/Asir 自体の性能限界を示すものではない。

```
#include <stdlib.h>
#include "appcalg.h"
main()
{
  randomize();                          // C の乱数初期化
  InitAppCAlg();                        // 初期化
  Real err,Q1,Q2,answer,x("x");         // 実数式の宣言
  double a,b;
  while(1)
  {
    a=(double)(rand()%20 - 9);          // C によって乱数 a,b を生成
    b=(double)(rand()%20 - 9);          // (C 言語の資産利用)
    Q1=(x+a);                           // 問題の 2 次式の 2 つの因子
    Q2=(x+b);                           // Q1,Q2 を生成
    cout <<"問題：右の 2 次式を因数分解せよ。 =>    "<< Q1*Q2; // 出題
    cout << "\n\n          答を入力して下さい。          =    ";
    if(!(cin >> answer)) break;         // 答を実数式 answer に入力
    err=Q1*Q2 - answer;                 // 正しいかどうかチェック
    if(err.is_zero()) cout<<"\n ★★★     正解です。   ★★★ \n\n\n";
    else cout <<"\n × 正解は: "<< Q1<<Q2 <<" です。 \n\n\n";
  }
  cout << "\n よくがんばりました。 \n";
}
```

Fig. 12.　因数分解練習ドリルプログラム

な複雑な場合には C 言語の持っている資産を利用できることは重要な意味を持つ。

4.4. 数値・数式融合処理の例

野田・佐々木氏の論文 [2] に基づいて近似 GCD 算法を実行してみた。目的は 1 変数多項式 p1, p2 の最大公約多項式を求めることである。その前にまず、ユークリッドの互除法について考察する。2 つの多項式 p1, p2 を受けて、ユークリッドの互除法により求めた最大公約多項式を返す関数 gcd() を Fig. 13 に示す。プログラム中の p2.is_zero() は p2 がゼロかどうかの判定であり、remainder(p1,p2) は p1 を p2 で割った余りを求める関数である。最後に return(primitive(p1)) で p1 の原始的部分を返す。ところがこの実行例 (Fig. 14) を見ると 3 番目の結果が正しくない。そこで 3 番目の処理について remainder() 関数を呼び出した結果を逐次表示してみると Fig. 15 のようになっている。すなわち、本来 3 回目のステップでゼロとなって終結すべきところが、浮動小数点数の演算誤差のために完全にゼロとはならなかったのである。したがって本数式処理モデルでは近似代数計算を考える必要がある。近似 GCD 算法に基づいて gcd() を書き直したものが Fig. 16 であ

```
Polynomial gcd(Polynomial p1, Polynomial p2)
{
  Polynomial r;
  while(! p2.is_zero())
  {
    r=remainder(p1,p2);
    p1=p2;
    p2=r;
  }
  return(primitive(p1));
}
```

Fig. 13. ユークリッドの互除法による GCD

```
gcd( (x+1)*(x+2)*(x+3) , (x+1)*(x+2) )    = [x^2+3*x+2]
gcd( (x+1)*(x+2)*(x+3) , (x+2)*(x-1) )    = [x+2]
gcd( (x+1)*(x+2)*(x+3) , (x+3)*(x-3)*x )  = -3.552714e-14
```

Fig. 14. ユークリッドの互除法による実行例

る。異なる点は第3引数に打ち切りパラメタ eps を指定できる (省略すると 1.0e-12 となる) ところである。アルゴリズムの詳細については文献 [2] および Fig. 16 のコメントに譲るが、上と同じ計算例 (Fig. 17) および近接根の場合の実行例 (Fig. 18) はいずれも期待どおりの結果を得ている。

このように、数値評価と数式処理が頻繁に繰り返されるような場合にもスムーズにプログラミングできることが分かる。しかも、数値計算部分が複雑になるほど威力を発揮するといえる。

5. むすび

以上、C++ による数式処理ライブラリーの実現方法について議論してきた。
ここで提案した C++ 数式処理モデルの特徴は次のようにまとめられる。

- 数式処理がコンパイラ型のプログラミングとして実行できる。
- 数値計算と数式処理の融合が容易である。
- 実行コードがインタプリタ型システムに比べてコンパクトで作業領域が広く取れる。
- C(++) 言語の持つ資産がそのまま利用できる。
- 数オブジェクトが double 型のみで数値計算の挙動が明快である。

しかし、C++ 数式処理を実現する上で次のような問題点が指摘される。

```
remainder => [6*x^2+20*x+6]
remainder => [1.111111*x+3.333333]
remainder => -3.552714e-14
remainder => 0
```

Fig. 15. 計算過程における余りの表示

```
Polynomial gcd(Polynomial p1, Polynomial p2, double eps=1.0e-12)
{
  double mq;
  Polynomial q,r;
  while(maxCoef(p2) > eps)    // p2 の係数の絶対値の最大値が eps 以下に
  {                           // なるまで処理を繰り返す
    divide(p1,p2,q,r);        // p1 を p2 で割った商 q と余り r を求める
    mq = maxCoef(q);          // 商 q の最大絶対係数を mq とする
    p1 = p2;
    if(mq > 1.2) p2 = r/mq;   // mq と 1.2 (マシン ε) を比較して規格化
    else p2 = r;
  }                           // 答が数なら 1 を返す
  if(p1.is_Number()) return(*new Polynomial(1.0));
  else return(primitive(p1));  // p1 の原始的部分を返す
}
```

Fig. 16. 近似 GCD 算法によるサブルーチン

```
App_gcd((x+1)*(x+2)*(x+3),(x+1)*(x+2) )    = [x^2+3*x+2]
App_gcd((x+1)*(x+2)*(x+3),(x+2)*(x-1) )    = [x+2]
App_gcd((x+1)*(x+2)*(x+3),(x+3)*(x-3)*x )  = [x+3]
```

Fig. 17. 同じ計算例

```
p1 = (x-0.5)*(x-0.502)*(x+1)*(x-2)*(x-1.5)
p2 = (x-0.501)*(x-0.503)*(x-1)*(x+2)*(x+1.5)
App_gcd(p1,p2,0.01)    = [x^2-1.001637*x+0.250818]
App_gcd(p1,p2,0.001)   = [x-0.501502]
App_gcd(p1,p2)         = 1
```

Fig. 18. 近接根を持つ場合の近似 GCD

- 新たにオーバーロードして利用できる演算子記号が C++ によって限定されていること。しかも、演算の優先順位も明確に決められていること。
  例えば、本モデルにおいてベキ演算記号として ^ を採用しているが、本来ビット演算に使われていたため優先順位の違いから、使用には明示的な括弧 () を利用するなどの注意が必要となる。

また、 コンパイラ型の実行手続きに従うことから、

- インタプリタ型数式処理システムに比べて実行に手間がかかり、即応性に欠ける。
  しかし、私の経験では、実際の研究に数式処理を応用する場合、あらかじめファイルにプログラムやデータを入力しておき、自動実行させるのがほとんどで、インタプリタとして利用する部分はほんのわずかなことが多い。その意味で C++ 数式処理の方向は十分実用になると考えられる。

これまで述べてきたように、四則演算レベルのモデルによって C++ からのアプローチがすぐれた特徴を持つことを見てきた。今後はさらに、実数式を要素とするリストやベクトルといったより高度な表現を考えた場合の方法と問題点について考察していきたい。

最後に、この論文をまとめるに当たり、いろいろとアドバイスを頂いた愛媛大学の野田松太郎氏に深く感謝いたします。また、 C++ 数式処理の動作を比較するに当たり、Risa/Asir の資料を提供して下さった富士通国際研の竹島卓氏、野呂正行氏および加藤昭彦氏に感謝いたします。

参 考 文 献

[1] 野田松太郎, 佐々木建昭, 鈴木正幸: 数式処理と数値計算の融合による精度保証, 情報処理学会論文誌, Vol. 31, No. 9, pp. 1204–1211 (1990)

[2] Matu-Tarow Noda, Tateaki Sasaki: Approximate GCD and its application to ill-conditioned algebraic equations, *J. Comp. App. Math.*, 38, pp. 335–351 (1991)

[3] 加藤昭彦, 野呂正行, 竹島卓: 数式処理システムｒｉｓａのパーソナルコンピュータへの移植について, 富士通国際研 (1991)

[4] Masayuki Noro: *Asir User's Manual*, 富士通国際研, Ver. 1.1 (1992)

[5] 下山武司: Asir コマンドマニュアル, 富士通国際研 (1992)

[6] Borland International: Borland C++ ユーザーズガイド, ボーランドジャパン (1991)

[7] Borland International: Borland C++ プログラマーズガイド, ボーランドジャパン (1991)

[8] R.S.Wiener, L.J. Pinson (今井慈郎, 宮武明義訳): C++ ワークブック, アジソン ウェスレイ・トッパン (1991)

[9] 吉田弘一郎: TURBO/BORLAND C++ によるオブジェクト指向狂詩曲, 技術評論社 (1992)

[10] 佐々木建昭, 元吉文男, 渡辺隼郎: 数式処理システム, 昭晃堂 (1986)